Detection of Violation Causes in Reflexion Models

Sebastian Herold, Michael English, Jim Buckley
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland
{firstname.surname}@ul.ie

Steve Counsell
Department of Computer Science
Brunel University London, UK
steve.counsell@brunel.ac.uk

Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin, Ireland
mel.ocinneide@ucd.ie

Abstract—Reflexion Modelling is a well-understood technique to detect architectural violations that occur during software architecture erosion. Resolving these violations can be difficult when erosion has reached a critical level and the causes of the violations are interwoven and difficult to understand.

This article outlines a novel technique to automatically detect typical causes of violations in reflexion models, based on the definition and detection of typical symptoms for these causes. Preliminary results show that the proposed technique can support software architects’ navigation through reflexion models of eroded systems to understand causes of violations and to systematically take actions against them.

Index Terms—software architecture erosion; architecture violations; architecture violation causes; reflexion modeling.

I. INTRODUCTION

The architecture of a software system manifests its most fundamental design decisions and therefore strongly affects the ability of the system to meet its functional and extra-functional requirements [1]. However, the software architecture of a system might suffer from effects known as software architecture erosion and architectural drift [2]. These effects may cause a progressive divergence between the “as intended” architecture and the “as implemented” architecture, leading to the gradual diminution of the system. Systems suffering from these phenomenon are unlikely to fulfil their desired quality attributes in the long run [3]. Many studies have observed that severe architecture erosion is prevalent in industrial software systems [4-8].

Reflexion Modelling [9] is one of the most successful techniques to address such architectural inconsistencies w.r.t industrial acceptance and application [4-7] as well as availability and maturity of tool support [10-14]. In reflexion modelling, differences between the intended software architecture and the actual architecture as implemented are automatically computed based on a manual mapping between architectural modules and units of source code. A reflexion model depicts these differences graphically.

Detecting and representing architectural violations in a reflexion model are only the first two steps towards resolving such violations and towards reversing the architecture erosion they represent. After that, violations have to be analyzed to understand what causes them in order to take effective actions to resolve the violations. This can include source code refactoring, modification of the mapping between code and architectural model, or modification of the architectural model itself.

As long as the number of violations is relatively low, and violations are largely unrelated, an experienced architect familiar with the system at hand will probably be able to perform these tasks manually: Reflexion modelling tools like JITTAC [14] provide the possibility to investigate which source code elements contribute to violations and to easily navigate to these elements, facilitating a quick case-by-case analysis of the existing violations.

However, a manual process is not feasible for systems that suffer from severe architecture erosion. It is not only the large number of violations but also the potentially complex relationships that their causes may have, such as violations having a joint cause or violations causing each other. Hence, the research presented in this article is driven by the following research question: is it possible to provide software architects with effective decision support in determining how to resolve multiple inter-twined architectural violations in reflexion models by automated analysis of violation causes?

As first step towards addressing this question, this article outlines a novel technique to automatically identify and rank causes of violations, based on the detection of so-called symptoms that are similar to anti-patterns or bad smells [15,16]. Furthermore, the article describes a preliminary evaluation of the approach in which the technique is applied to guide the software architect through the violations of an eroded system.

II. PROPOSED TECHNIQUE

In this section, we first introduce the basic terminology of reflexion modelling. Then, the three main components are explained in the order they need to be defined and applied: identification of violation causes, definition of indicative symptoms for violations, and detection and comparison of violations causes in reflexion models based on the detection of symptoms. The section is concluded with an example of a violation cause and a description of a tool prototype.
A. Foundations and Terminology of Reflexion Modelling

Reflexion Modelling was originally introduced to check the difference between the higher-level design of software systems and their implementations in a light-weight and iterative manner [9]. Applying this technique can be separated into three steps. Firstly, a high-level model of the system’s structure is created by the user, consisting of architectural modules and relations between these architectural modules, indicating expected dependencies. Secondly, units of the source model are manually mapped to the architectural modules, creating an abstraction of the implementation and expressing which parts of the source code implement which modules. Finally, a reflexion model is automatically computed by comparing the expected dependencies between architectural modules with the actual dependencies between the source model units implementing the modules. Three different forms of relationships between expected and actual dependencies are visualized in different ways (Fig. 1): convergences indicating that actual dependencies are aligned to expected dependencies; absences indicating that expected dependencies between architectural modules are not present; and divergences indicating actual dependencies that are undesired or unexpected. We call source code relationships contributing to a divergence architecture violations.

B. Common Causes of Violations in Reflexion Models

Several case studies in applying architecture consistency checking based on reflexion modelling-based approaches have analysed causes for architectural violations [4,7,8]. The results suggest that there are a couple of recoucurring common causes, identified in different commercial software systems.

In this work, we focus on two of the most common causes identified in those studies. Architecturally misplaced software units occurs when the source or target element of a source code relationship contributing to a divergence is in the wrong module, i.e. it is an element of the module it is mapped to but realizes functionality that does not match the module’s purpose. For example, a class implementing a utility function that is mapped to the GUI module in Fig. 1 would constitute a misplaced class. Although violations caused by this are often considered “trivial” [7] and of having minor short-term impact on the functional correctness of the system, they may negatively affect quality attributes such as modularity, adaptability and robustness.

The second common cause is related to callbacks which are violations in which a convergence between two modules A and B, there is a divergence between B and A caused by callback mechanisms reacting to the architecturally consistent main control flow directed from A to B, e.g., listener/observers in the GUI that are invoked by the application core. Often only the dependencies required for the main direction of control flow are covered by a convergent edge in the reflexion model of a system. Dependencies required for callbacks are considered allowed exceptions contributing to divergences [8]. We call this violation cause architecturally divergent callbacks.

C. Symptoms of Violation Causes

In order to provide tool support for the automated detection of violation causes, we need algorithmically computable indicators for these causes. These indicators are called symptoms here as this is in line with the general definition of this term: “an indication of the existence of something, especially an undesirable situation” [17].

We distinguish two different types of symptoms. First, structural symptoms describe structural patterns in the reflexion model, the mapping, and the code, that might indicate a certain violation cause. For example, the required invocation relationship combined with the divergence/convergence relationships as described in the previous section form the pattern for the symptom indicating callback-like relationships.

The second type of symptoms are measurable symptoms which describe quantifiable properties of the elements in structural symptoms. Measurable symptoms are expressed in terms of metrics and desirable target values that assess the existence of the symptom. For example, metrics measuring the relative number of violations a class is involved in, or how similar its actual usage relationships are compared to what its architectural module allows, can be applied to indicate a misplaced class. In order to compare values for different violation causes, we only allow metrics yielding values from a bounded interval (see Sec. II.D).

D. Evaluation and Comparison of Violation Causes

The general evaluation of the probability that a known violation cause is the actual cause for a specific violation first detects whether the specified structural symptom(s) match. Then, the vector of metric values as specified in the measurable symptoms is computed for the entities in the specific occurrence of the structural pattern. Then, the Euclidian distance between this vector and the vector consisting of the reference values specified by the violation cause is calculated and divided by the maximal distance (which is defined since all metrics values are from a bounded interval).

The shorter the resulting distance is, the more likely the general cause is the actual cause of a specific violation. Hence, the approach is capable of expressing uncertainty regarding the relationship between violations and potential violation causes and to rank different violation cause candidates.

E. Example of a Violation Cause and its Symptoms

Misplaced software units can occur on the level of classes and interfaces meaning that a complete class can be inappropriately mapped to an architectural module. However, it might instead be the case that the class is in the appropriate
module but a single field or method is inappropriately placed. This can happen if this field or method does not support the class’ prime functional concern but is part of some other functionality that should not be realized in the architectural module the class is in.

Table I illustrates the relevant symptoms for this violation cause. The figure shows a method $f$ in class $C$ in a module $M$ is the source of a violation towards $D$ by using $D$ for typing parameters or local variables, accessing $D$’s fields, or calling $D$’s methods. This describes a structural symptom. The second symptom is a low $\text{modsim}(f)$ value, to indicate that $f$ is misplaced in $M$. This metric combines two metrics $\text{relviol}(f)$ and $1 - \text{convsim}(f)$ in a weighted sum. $\text{relviol}(f)$ counts the ratio of violations that $f$ is involved in to all relations incoming to and outgoing from $f$, and is hence a value in the range [0,1]. A high value might indicate wrong placement. $\text{convsim}(f)$ compares the set of modules that $f$ actually depends on with the set of modules it would be allowed to used according to the convergences modelled in the architectural model. A low value - values are in the range [0,1] - might indicate that $f$ is placed in the wrong module.

The metric $\text{modsim}(f)$ results in a value in the range [0,1] with a high value indicating a high probability of $f$ being misplaced. It can be computed for any source code elements such as classes to evaluate misplacement of functionality at different levels of granularity. The fourth symptom shown in Table I, suggests that the class $C$ is in the right place indicated by a low value for $\text{modsim}(C)$. The third symptom uses $\text{cohesion}(f,C)$ to express a low relatedness between $f$ and $C$ based on cohesiveness; this metric is an adaptation of the TCC metric [18]. It measures cohesion based on the density of a graph in which edges represent pairs of methods in a class that are related because they access the same attribute.

F. JITTAC Medic
The previously described concepts have been integrated into the JITTAC reflexion modelling tool for Eclipse [14]. This extension, called JITTAC Medic, computes the likeliness of the previously introduced violation causes for the violations occurring in a reflexion model. Users of the tools can access this information in two different ways in order to understand or to subsequently resolve the violations.

In case the user is investigating the cause of a single violation in particular, JITTAC Medic can show him/her a ranked list of potential causes per violation. The user simply selects a divergence in the reflexion model, and JITTAC shows a list of all violations in the source code contributing to that divergence. The user can right-click on an entry in this list, bringing up a list of possible violation causes sorted by the distance between the actual metric values and the specified symptoms as described above.

Another option is particularly useful if the number of violations or the complexity of the system at hand is too large to decide which violations should be investigated. JITTAC Medic can list the top five violations in the overall reflexion model ranked by the likelihood that any of the known general violation causes is the actual cause for those violations. The motivation of this list to guide the user through the model by drawing his attention towards violations that have clear causes and that can hence be more easily resolved than those for which the cause might only become clearer after manual inspection- which might be easier after the total number of violations has been reduced.

III. PRELIMINARY EVALUATION
Here, we investigate the outlined approach to address architecture erosion in an open source application.

A. The Investigated System: JMoney
We selected the open source private finance tracker software JMoney (version 0.4.4) for our application scenario. It is suitable for a preliminary evaluation of the proposed approach for several reasons. It is sufficiently complex to be a candidate for an eroded system on the one hand but small enough to have the proposed violation causes checked manually on the other. Furthermore, it has been investigated by Shah et al. regarding cycles in the package structure of the source code [19]. This package structure can serve as an intended software architecture which is valuable in the world of open source software where architectural documentation is often lacking [20].

We translated the package structure, which Shah et al. created from removing cycles, into an architectural model in JITTAC, reflecting their hypothesized intended software architecture [19]. Mapping the original, non-refactored code base to the model, resulted in a reflexion model showing the divergences as depicted in Fig. 2. While only class moves were applied in [19] to resolve cycles, we were interested in other violation causes that JITTAC Medic could identify, resulting in a possibly differently refactored system structure. More

<table>
<thead>
<tr>
<th>Violation Cause</th>
<th>Architecturally Misplaced Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A method $f$ refers to a class/interface $D$, accesses one of its fields, or calls one of its methods, causing a violation, but $f$ neither belongs to its containing class $C$ nor to the surrounding module $M$.</td>
</tr>
<tr>
<td>Symptom</td>
<td>$\text{modsim}(f) = 0$</td>
</tr>
<tr>
<td>Symptom</td>
<td>$\text{cohesion}(f,C) = 0$</td>
</tr>
<tr>
<td>Symptom</td>
<td>$\text{modsim}(C) = 1$</td>
</tr>
</tbody>
</table>
specifically, we are interested to see whether being guided by the top-five recommendations of JITTAC Medic allows us to systematically repair the system.

B. Violation Causes in JMoney

Following the list of the most likely violation causes, JITTAC Medic draws attention to the Start class in its ranked list. It is part of the “app” module, containing the core application logic, but initializes the main GUI component, causing violations towards the “gui” module. JITTAC Medic recognizes it as a “Misplaced Class” with distance 0 to the symptoms of the violation cause.

The next violation cause, that the list suggests, is a case of “Misplaced Class” for the class AccountChooser, which is currently mapped to “gui”. This is justified by the fact that is mainly used by IO functionality located in “io”. Hence it seems to be more appropriate there. However, the class mixes GUI and IO functionality leading, making the correct module hard to decide since the separation of concerns is not clear-cut. We hence chose to reject the suggested violation cause, after we had tentatively mapped it to “io”, creating new violations.

The next violation cause suggests that SortedTreeModel, mapped to “app” was misplaced (distance 0.321), and after moving it to “model” as being part of the data model, that SortedTreeNode should be replaced as well (distance 0.214). After that, JITTAC Medic suggests that the method fileReadError(...) is used as a callback. It is located in “gui” as part of the class Mainframe and is called by classes in “io” (distance 0.333). This method is indeed used for notifying and presenting a dialogue to the user if the import of data fails. An instance of Mainframe is registered with the import classes in “io”, so they can callback via its fileReadError(...) method. We chose to mark the dependencies between “io” and “gui” related to this as exceptions in the reflexion model.

The next violation cause suggests that the NEW attribute of the class Constants in “app” is misplaced due to its usage in “io” and “model” (distance 0.422). However, other fields of the same class show similar values, and there is an indication that the complete class might be incorrectly placed. Moreover, it must be stated that a class like Constants which, as a collection of constants, is not cohesive in the object-oriented understanding as applied in the symptoms for misplaced methods/attributes. Nevertheless, it might be useful to keep system constants in a single class. Hence, we chose to address the violations related to this by considering the whole class as misplaced and by moving it to “model”. The same action is taken for Currency, which is located in “app” but should be part of “model” according to its functionality (considered a misplaced class with distance 0.438).

After this step, only twelve violations in the reflexion model remain which are all related to AccountChooser. Addressing the cause of these violations, which lies in the incoherent design of the class as described above, would probably require more complex restructuring, such as splitting the class and is beyond the scope of this preliminary evaluation.

C. Discussion of Results

Most of the classes moved in this evaluation were also moved in the approach to remove dependency cycles presented in [19]. The only exceptions are SortedTreeModel and SortedTreeNode which are not mentioned in [19]. In that study also classes of “io” were moved to “gui” in order to remove the dependency cycles between the packages corresponding to these modules. The approach proposed here, however, identified callbacks between “io” and “gui” as the reason for this cycle, which should not be resolved by moving import/export functionality to a module responsible for presentation functionality. This has been correctly discovered by JITTAC Medic. As a consequence of moving import/export functionality to “gui”, the approach in [19] does not recognize the AccountChooser class as a problem (all dependencies involving this class are local to “gui” after this operation).

The Constant class showed that the symptoms for misplaced members probably have to be refined, in particular the way cohesion is measured. While the implemented metric works well for classes that are well-designed in an object-oriented sense, it needs to be adapted for data types and possibly other forms of types.

IV. Related Work

To our best knowledge, there are no similar approaches to the analysis and automated detection of violation causes in reflexion modelling. Although the detection of code smells or anomalies has been intensively investigated in the last decades (overviews can be found in [21,22]), an integration with reflexion modelling is missing.

The most closely related research to the proposed approach w.r.t. the analysis of the relationship between architectural violations and their causes in code, is a body of work that has been conducted by a group of authors in recent years [23-25]. They showed that a statistically significant cause-effect relationship between code anomalies and architectural problems exists [23]. They adapt object-oriented design metrics to consider information about how code elements were mapped to architectural components and concerns [24]. This minimal architectural information should enable the detection of code anomalies in the absence of an explicitly specified software architecture.

The most important conceptual difference in the proposed approach is the assumption that a specification of the intended software architecture exists or will be created through reflexion modelling. As case studies have shown, this is a feasible and useful task in industrial practice [4,7]. This also means that the proposed approach can exploit existing knowledge about architectural violations and relationships in source code as specified in the structural symptoms of violations causes. We assume that in the scenario of an existing architectural description, this will lead to a more accurate recommendation of violation causes.

Another important difference lies in how code anomalies/symptoms are computed. The approach described by Macia et al. [24] is based on detection strategies [26], which are rules made of metrics, thresholds and operators to compose strategies, leading to a binary decision whether an architecturally relevant code anomaly has been found. In contrast the proposed approach is able to express uncertainty in the cause-effect
relationships of violations and to compare different causes, allowing the software architect to perform a fine-grained analysis of architectural violations.

V. CONCLUSION

As outlined in Sec. I, the main question motivating this research is whether automated detection of violation causes in reflexion models can effectively support software architects making decisions regarding the resolution of software architecture erosion. Despite the limitations of the preliminary case study, in which we have only investigated a single system so far, results are promising. The actions taken based on the violation causes ranked highest (based in turn on the specified symptoms) lead to reasonably restructured system that mirrors a previous restructuring effort to a large degree and improves it in some points. The consideration of violations causes, which aggregate information about a set of violations in context, can effectively guide the software architect through the reflexion model of an eroded system.

However, ongoing work on this approach will improve and extend the existing concepts and tools. A first extension will improve the current recommendation of violation causes which is currently based on locally optimal ranking w.r.t. single violations. We will investigate how to extend this in order to recommend a set of violation causes for the violations in a system that is optimal w.r.t to global criteria such as minimal number of causes or highest mean similarity to symptoms. Furthermore, we will investigate which information and forms of presentation in addition to the list of top-ranked violations causes might be required by architects to make adequate decisions.

Moreover, we will review our existing empirical studies and conduct new studies on how reflexion modelling is used in industrial practice and which violation causes can typically be found to further extend the set of supported violation causes and the capabilities of the developed tool support so far. This also includes the selection and definition of adequate metrics, as well as their refinement and weighting, to measure properties of reflexion models and the related source code.

Furthermore, we will extend the approach towards support to automatic recommendations of repair actions to resolve architectural violations. Due to the combinatorial complexity of the problem of which refactorings to apply in order to repair an eroded system [27], techniques like heuristic search seem promising. They could benefit from the reduction of the problem space that the aggregation of violations by detecting potentially joint causes provides.

ACKNOWLEDGMENT

This work was supported, in part, by Science Foundation Ireland grants 10/CE/I1855 and 12/IP/1351 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

REFERENCES


